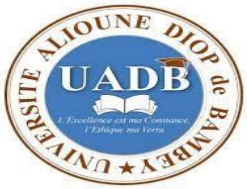
REPUBLIQUE DU SENEGAL

Un Peuple – Un But – Une Foi

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR DE LA RECHERCHE ET DE L'INNOVATION

UNIVERSITE ALIOUNE DIOP DE BAMBEY



U.F.R DES SCIENCES APPLIQUEES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION(SATIC)
DEPARTEMENT TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION (TIC)
MASTER II SI

Ratelimiting: cas avec Express Js

Réalisé par:

Maty Sylla

Sous la direction de :

M . Maodo DIOP

Année Académique 2024 /2025

Introduction

Le *rate limiting* (limitation de débit) est un mécanisme essentiel dans les architectures web modernes. Il permet de contrôler et de restreindre le nombre de requêtes qu'un utilisateur, une adresse IP ou un service peut envoyer à une API durant une période donnée. Ce contrôle est indispensable pour garantir la qualité de service (QoS), protéger l'application contre les abus (comme les attaques par force brute ou les dénis de service), et assurer la stabilité ainsi que la disponibilité du système.

Dans le cadre de ce projet, nous présentons l'implémentation du *Limitation de taux* Dans une application **Node.js** Utilisation du cadre LE **Express.js**. Nous explorons deux approches :

1. **L'utilisation d'une bibliothèque existante** (), simple à mettre en place et adaptée aux besoins courants.express-rate-limit
2. **La création d'un middleware personnalisé**, permettant une maîtrise fine du fonctionnement interne du mécanisme.

Cette étude offre ainsi une compréhension globale du *rate limiting*, de ses enjeux, et de sa mise en œuvre pratique dans un environnement Node.js.

1. **Utilisation de la limite de débit express**

   express-rate-limit est une bibliothèque populaire qui simplifie la mise en œuvre de la limitation de débit dans les applications Express. Expliquons comment la mettre en place.

   Pour commencer, installez express-rate-limit en utilisant npm :

```
npm install express-rate-limit
```

   Voici une implémentation basique de la limitation de débit dans une application Express :

```
const express = requere('express') ;
const rateLimit = require('express-rate-limit') ;
const app = express() ;

 Configurez le limiteur de débit : maximum de 100 requêtes par 15 minutes par IP
const limiter = rateLimit({
 windowMs : 15 * 60 * 1000,  15 minutes
 max : 100,  limitez chaque IP à 100 requêtes par « fenêtre » (ici, toutes les 15 minutes)
 message : « Trop de demandes depuis cette IP, veuillez réessayer après 15 minutes »,
}) ;

 Appliquer le limiteur de vitesse à toutes les requêtes
 app.use(limiter) ;

app.get('/', (req, res) => {
 res.send('Bonjour, monde !') ;
}) ;

app.listen(3000, () => {
 console.log('Serveur fonctionnant sur http://localhost:3000');
});
```

Explication:

- windowM : Définit la fenêtre temporelle pendant laquelle les requêtes sont comptées (dansce cas, 15 minutes).
- max : Spécifie le nombre maximal de requêtes autorisées dans la fenêtre de temps (100 requêtes).

- message : Un message personnalisé retourné à l'utilisateur lorsque la limite de débit est dépassée.Vous pouvez appliquer le limiteur de vitesse globalement ou le restreindre à des itinéraires spécifiques. Par exemple, appliquer la limitation de taux uniquement aux routes API

```
app.use('/api/', limiter);
```

Cela protège vos points de terminaison API contre la subcharge des requêtes.

2. **Créer votre propre middleware limitant**

Pour plus de contrôle ou de personnalisation, vous pouvez créer votre propre middleware limitant la fréquence sans dépendre de bibliothèques externes. Cela vous permet d'affiner la logique selon les exigences de votre application.

a. Limiteur de débit personnalisé en mémoire

L'implémentation la plus simple consiste à stocker les comptes de requêtes en mémoire. Cette méthode est efficace pour de petites applications mais ne s'adapte pas bien à plusieurs instances. Voici un exemple de base :

```
const express = requer('express') ;
const app = express() ;

 Compte de requêtes de stockage par IP
const requêteCounts = {} ;

 Middleware
 custom rate limiter const rateLimiter = (req, res, next) => {
 const ip = req.ip ;
  const now = Data. maintenant() ;

if ( !requestCounts[ip]) {
requestCounts[ip] = { count : 1, lastRequest : now } ;
} else {
const timeSinceLastRequest = now - requestCounts[ip].lastRequest ;
temps const = 15 * 60 * 1000 ;  15 minutes

si (timeForceLastRequest < timeLimit) {
requestCounts[ip].compte += 1 ;
} sinon {
requestCounts[ip] = { count : 1, lastRequest : now } ;  Réinitialisation après fenêtre
```

```
 temporelle }
 }

 const maxRequests = 100 ;

 if (requestCounts[ip].count > maxRequests) {
 return res.status(429).json({ message : 'Trop de demandes, veuillez réessayer plus tard.' }) ;
 }

 requestCounts[ip]. lastRequest = maintenant ;
 next() ;
} ;


 Appliquer le limiteur
de vitesse personnalisé app.use(rateLimiter) ;

app.get('/', (req, res) => {
 res.send('Bienvenue sur le serveur à débit limité') ;
}) ;

app.listen(3000, () => {
 console.log('Serveur fonctionnant sur http://localhost:3000');
});
```

Explication

- Le middleware suit le nombre de requêtes de chaque adresse IP et l'heure de la dernière requête.
- Si le nombre de requêtes dépasse 100 dans les 15 minutes, le middleware bloque les requêtes supplémentaires en renvoyant un code d'état 429 Trop de requêtes.
- Le comptage se réinitialise après le passage de la fenêtre temporelle.

Avantages : Contrôle total de la logique.

Inconvénients : Le stockage en mémoire n'est pas évolutif pour les systèmes distribués ou les instances de serveurs multiples.

b. Stockage des comptes de requêtes dans Redis

Pour les applications plus larges, stocker les comptes de requêtes dans Redis est une solution plus évolutive. Redis est un stockage de données rapide en mémoire qui peut être utilisé pour suivre le nombre de requêtes à travers différentes instances de votre application.

Pour commencer avec Redis, vous devrez installer les packages redis et ioredis :

```
npm install redis ioredis
```

Ensuite, vous pouvez modifier votre middleware limitant le débit pour stocker le nombre de requêtes dans Redis :

```
const express = requer('express') ;
const Redis = require('ioredis') ;
const app = express() ;

const redis = nouvelle Redis() ;

 Limiteur de débit utilisant Redis
const rateLimiter = asynchrone (req, res, next) => {
 const ip = req.ip ;
  const currentTime = Date. maintenant() ;
 clé const = 'limite-débit :${ip}' ;

 limite de const = 100 ;  Max requests
 const windowTime = 15 * 60 ;  15 minutes en secondes

 demandes de const = attendre les redis. incr (clé ) ;

 si (requêtes === 1) {
  Réglez l'expiration de la clé à la fenêtre temporelle de la première requête
 await redis.expire (clé, windowTime) ;
 }

 if (requests > limit) {
 return res.status(429).json({ message : 'Trop de demandes, réessayez plus tard.' }) ;
 }

 suivant() ;
} ;

 Appliquer le limiteur de vitesse à l'API routes
 app.use('/api', rateLimiter) ;

app.get('/api', (req, res) => {
 res.send('API limitée en débit') ;
}) ;

app.listen(3000, () => {
```

```
 console.log('Serveur fonctionnant sur http://localhost:3000');
});
```

Comment cela fonctionne :

- La clé Redis est l'adresse IP de l'utilisateur. Redis augmente le nombre de requêtes à chaque appel API effectué depuis cette IP.
- L'expiration de la clé est fixée à la fenêtre de temps définie (15 minutes), et la clé se réinitialise automatiquement après cette fenêtre.
- Si le nombre de requêtes dépasse la limite, le serveur renvoie un code d'état 429.

  Avantages : S'adapte bien aux applications distribuées.

  Inconvénients : nécessite une configuration Redis et une configuration supplémentaire.

  Cas d'utilisation pour la limitation de taux

  • Protection des API publiques : Les API publiques sont très susceptibles d'être abusées et peuvent subir un trafic important de la part d'utilisateurs non autorisés ou malveillants. La limitation des taux permet d'assurer une utilisation équitable et évite qu'un seul utilisateur ne consomme des ressources excessives.

  • Réduction de la charge serveur : En limitant le nombre de requêtes qu'un client peut effectuer, vous pouvez protéger votre serveur contre une surcharge excessive de requêtes, ce qui pourrait dégrader les performances pour d'autres utilisateurs.

  • Prévention des attaques par force brute : La limitation de taux est efficace pour ralentir les tentatives de connexion par force brute en limitant le nombre de tentatives de connexion en une courte période.

Conclusion

Ce projet a permis d'explorer trois approches simples de rate limiting avec Express.js : la bibliothèque express-rate-limit, le middleware personnalisé, et Redis. Ces techniques sont essentielles pour garantir la sécurité, la performance et la qualité de service des API modernes.